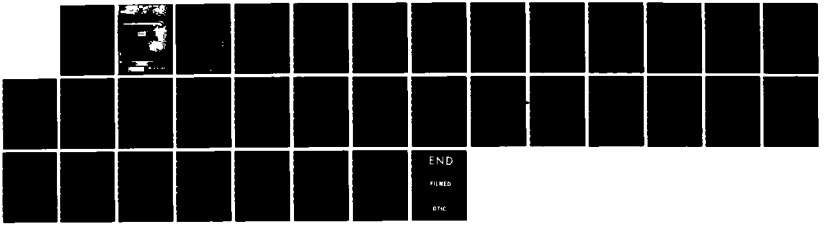
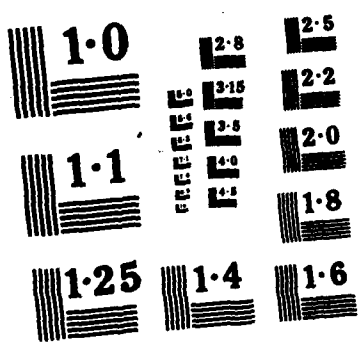
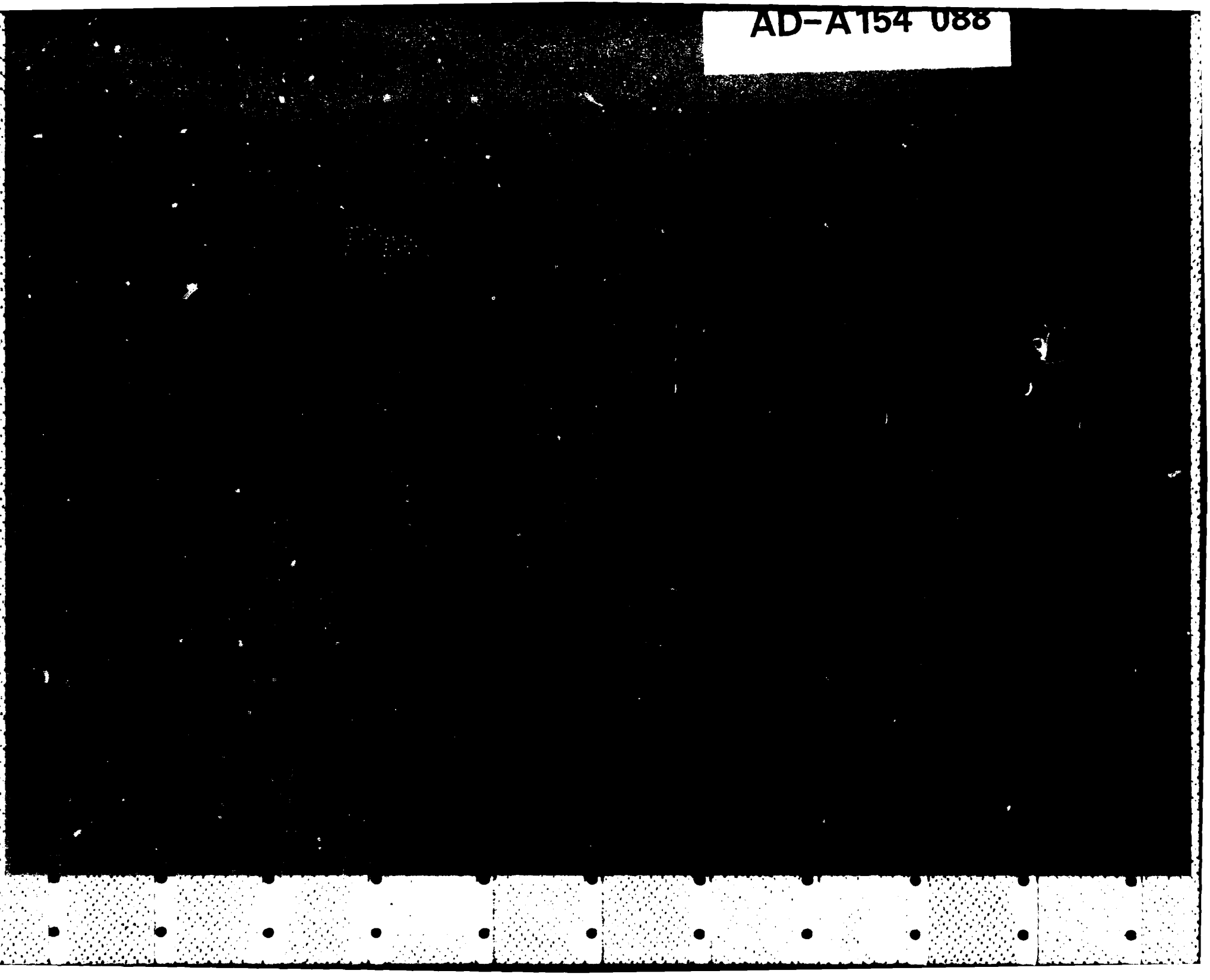
AD-A154 088 NESTED DISSECTION ON A MESH-CONNECTED PROCESSOR ARRAY (U) STANFORD UNIV CA CENTER FOR LARGE SCALE SCIENTIFIC COMPUTATION P H WORLEY ET AL. MAR 85 CLASSIC-85-08 1/1

UNCLASSIFIED N00014-82-K-0335 F/G 9/2 NL




1·0
1·1
1·25
1·4
1·6
2·8
3·15
3·5
4·0
2·5
2·2
2·0
1·8



Nested Dissection on a Mesh-Connected Processor Array

by

Patrick H. Worley*
Department of Computer Science
Stanford University, Stanford, CA 94305

and

Robert Schreiber
Guiltech Research Corporation Inc.
255 San Geronimo Ct.
Sunnyvale, CA 94086

This work has been submitted for publication in the *Proceedings of the ARO Workshop on New Computing Environments: Parallel, Vector, and Systolic.*

* This work was supported by the Center for Large Scale Scientific Computation, funded by Office of Naval Research Contract N00014-82-K-0335.

"Nested Dissection on a Mesh-Connected Processor Array" by Patrick H. Worley* and Robert Schreiber**

square Root of N

Abstract. We present a parallel implementation of Gaussian elimination without pivoting using the nested dissection ordering for solving $Ax = b$ where A is an $N \times N$ symmetric positive definite matrix. If the graph of A is a $\sqrt{N} \times \sqrt{N}$ finite element mesh then Birkhoff and George and Liu have shown that a parallel complexity of $O(\sqrt{N})$ can be achieved for Gaussian elimination with the nested dissection ordering. Our implementation achieves this parallel complexity on a two dimensional MIMD processor array with N processors and nearest neighbors interconnections. Thus nested dissection is a near optimal algorithm for this problem on this interconnection topology. The parallel implementation on this architecture requires $158\sqrt{N} + O(\log_2(\sqrt{N}))$ parallel floating point multiplications. It is faster than a Kung-Leiserson systolic array for banded matrices for $N \geq 961$, and faster than a serial implementation for N as small as 9.

Log of square root of N to base 2

* Department of Computer Science, Stanford University, Stanford, CA 94305.
This work was supported by the Center for Large Scale Scientific Computation, funded by Office of Naval Research Contract N00014-82-K-0335.
** Guiltech Research Corporation Inc., 255 San Geronimo Ct., Sunnyvale, CA 94086.

1. Introduction. With the advent of research into parallel computer architectures numerical algorithms must be reexamined in terms of their parallel complexity. A realistic parallel complexity analysis takes the communication cost into consideration, so a machine architecture must be assumed. There are two approaches form this point. Given a computer architecture the parallel complexity for competing algorithms on that architecture can be compared. With the introduction of VLSI technology and new design tools special purpose computers can also be economically feasible. Then the comparison is between the computational complexity of (algorithm, computer architecture) pairs for solving a given problem. This latter analysis need not be restricted to special purpose computers. Consideration of the mix of problems to be solved on a more general purpose parallel architecture, and the computational complexity of these (algorithm, architecture) pairs, may help indicate a cost effective parallel architecture.

Consider the problem of solving the linear system $Ax = b$, where A is a symmetric positive definite $N \times N$ matrix. We define the graph of A to be an undirected graph $G(A) = (V, E(A))$ where $V = \{1, 2, \ldots, N\}$ and $E(A) = \{(i,j) | a_{ij} \neq 0\}$. Let the graph of A be isomorphic to a $\sqrt{N} \times \sqrt{N}$ grid where each vertex has edges to its eight nearest neighbors, a *finite element mesh*, or to a subset of such a graph. See Figure 1. This is the model equation used by George [3] to describe the nested dissection algorithm. Such problems arise when solving a second order elliptic partial differential equation in two space dimensions. The graph of the matrix reflects a uniform finite difference or finite element grid in the spatial domain, or a simple mapping of such, with a differential operator approximation that only involves nearest neighbors. Finally, let $n \stackrel{\text{def}}{=} \sqrt{N}$ and restrict it to the form $n = 2^l - 1$ for some $l \in \{1, 2, \ldots\}$. This model problem is used to describe the algorithm, but our approach can easily be extended to handle more general problems.

Our goal is to introduce a computer architecture and a mapping onto this architecture which achieves the parallelism predicted by Birkhoff and George [1], and by Liu [10]. Their results give a parallel complexity of $O(n)$ using $O(n^2)$ processors for the matrix factorization $A = LDL^t$ and the upper and lower triangular matrix solves. By taking into account the communication time and the time required for program and data loading and unloading, a complete parallel complexity analysis can be made. We argue for the usefulness of this approach by emphasizing the generality and simplicity of the computer architecture, and the optimality of the nested dissection algorithm for this architecture. An algorithm similar to ours has been presented by Gannon [2]. He uses a blend of Givens rotations and block Gaussian elimination to triangularize the matrix (not factor it), and has somewhat different architectural requirements. His algorithm also involves larger constants and a more complex program. But some of the ideas are the same, and this work provides a detailed look at an implementation of these common ideas.

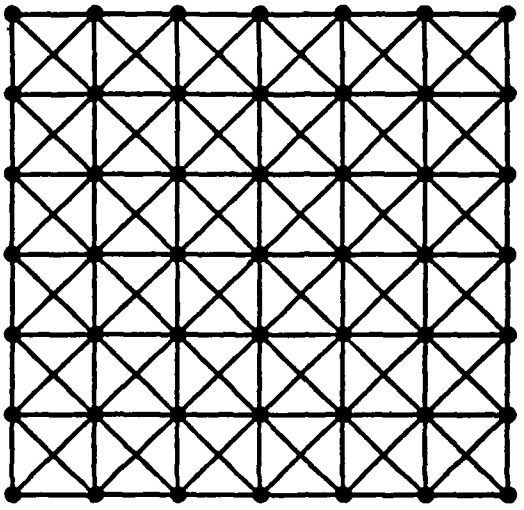


Figure 1: 7 by 7 finite element mesh.

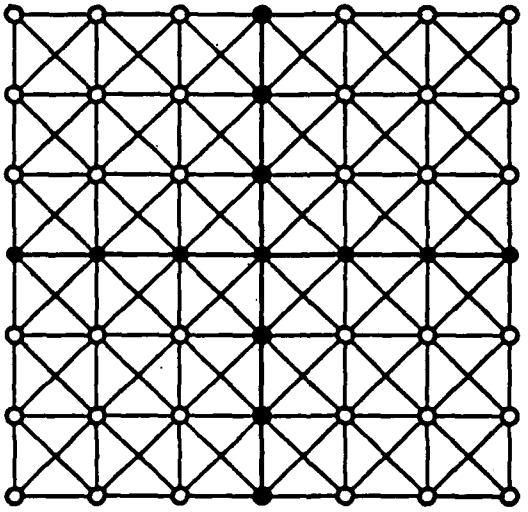


Figure 2: A cross of vertices and its four subgrids.

In sections 2 and 3 we review the symmetric factorization and the nested dissection algorithm. In section 4 we introduce our architecture and how to use it efficiently to solve subproblems arising in the nested dissection algorithm. Section 5 describes the parallel factorization algorithm. Section 6 describes the parallel upper and lower triangular matrix solves. Section 7 discusses practical aspects of using the algorithm, and section 8 analyzes the efficiency of the algorithm.

2. The Symmetric Factorization. We solve $Ax = b$ by calculating a symmetric factorization

$$A = LDL^t$$

with L unit lower triangular and D diagonal, and forward and backward triangular matrix solves

$$Lz = b, \qquad L^t x = D^{-1} z.$$

Let I_r be the $r \times r$ identity matrix. Let I represent identity matrices whose sizes are clear by context. Define $A^{(0)} = A_0 = A$ and $L^{(0)} = I_N$. Assume we have a factorization

$$A = L^{(i-1)} A^{(i-1)} L^{(i-1)^t}$$

for $i \geq 1$ where $L^{(i-1)}$ is unit lower triangular and $A^{(i-1)}$ is block diagonal:

$$L^{(i-1)} = \begin{pmatrix} L_{i-1} & \\ C_{i-1} & I \end{pmatrix}, \qquad A^{(i-1)} = \begin{pmatrix} D_{i-1} & \\ & A_{i-1} \end{pmatrix}.$$

L_{i-1} is an $(i-1) \times (i-1)$ unit lower triangular matrix, and D_{i-1} is an $(i-1) \times (i-1)$ diagonal matrix. C_{i-1} is an $(N-i+1) \times (i-1)$ matrix. A_{i-1} is an $(N-i+1) \times (N-i+1)$ matrix. Label the elements of A_{i-1},

$$A_{i-1} = \begin{pmatrix} d_i & \bar{u}_i^t \\ \bar{u}_i & \tilde{A}_i \end{pmatrix},$$

where d_i is a scalar and $\bar{u}_i$ is a vector of length $N-i$.

$A^{(i-1)}$ can be factored,

$$A^{(i-1)} = \begin{pmatrix} I & & \\ & 1 & \bar{0}^t \\ & \frac{1}{d_i}\bar{u}_i & I \end{pmatrix} \begin{pmatrix} D_{i-1} & & \\ & d_i & \bar{0}^t \\ & \bar{0} & A_i \end{pmatrix} \begin{pmatrix} I & & \\ & 1 & \frac{1}{d_i}\bar{u}_i^t \\ & \bar{0} & I \end{pmatrix},$$

where

$$A_i = \tilde{A}_i - \frac{\bar{u}_i \bar{u}_i^t}{d_i} \tag{2.1}$$

and $\bar{0}$ is the zero vector of length $N-i$.

Thus the factorisation of A becomes

$$\begin{pmatrix} L_{i-1} & \\ C_{i-1} & I \end{pmatrix} \begin{pmatrix} I & \\ & \begin{matrix} 1 & \bar{0}^t \\ \frac{1}{d_i}\bar{u}_i & I \end{matrix} \end{pmatrix} \begin{pmatrix} D_{i-1} & \\ & \begin{matrix} d_i & \bar{0}^t \\ \bar{0} & A_i \end{matrix} \end{pmatrix} \begin{pmatrix} I & \\ & \begin{matrix} 1 & \frac{1}{d_i}\bar{u}_i^t \\ \bar{0} & I \end{matrix} \end{pmatrix} \begin{pmatrix} L_{i-1}^t & C_{i-1}^t \\ & I \end{pmatrix}$$
$$\equiv \begin{pmatrix} L_i & \\ C_i & I \end{pmatrix} A^{(i)} \begin{pmatrix} L_i^t & C_i^t \\ & I \end{pmatrix}.$$

L_i is an $i \times i$ unit lower triangular matrix, and C_i is an $(N-i) \times i$ matrix.

Since A is symmetric positive definite, d_i is nonzero, and this process describes an algorithm to produce the symmetric factorization.

3. Nested Dissection. When only a small fraction of the elements of A are nonzero, A is said to be sparse. When a sparse matrix is factored, *fill-in* can occur: the triangular factors contain nonzeroes in positions where A has zeroes. An increase in the number of nonzeroes increases the work in computing the factorisation and the triangular matrix solves. The amount of fill-in is sensitive to the ordering of the columns of the matrix, also called the variable ordering.

Nested dissection is a symmetric factorisation $A = LDL^t$ with a particular heuristic for the variable ordering intended to minimise fill-in [4]. At least $n^3/6 + O(n^2)$ multiplications are required to factor A for our model problem [3][6]. The nested dissection variable ordering leads to a multiplicative operation count of $267/28\, n^3 + O(n^2 \log_2 n)$, and so is close to optimal.

The symmetric factorisation can be described in terms of the graph of the matrix A. Each vertex of the graph represents a variable or column of the matrix, and each edge between the vertices v_i and v_j represents a nonzero a_{ij} or a_{ji}. Thus the graph of the matrix describes the sparsity pattern. The graph of the matrix A_i, produced by the i^{th} step of the symmetric factorisation (see equation (2.1)), is a simple modification of the matrix graph for A_{i-1}: remove the vertex associated with the i^{th} variable and add edges between all vertices that had been connected to it, if they don't already share an edge. The addition of edges reflects additional nonzeroes in the triangular factor L. We denote both the graph manipulation and the step in the matrix factorisation by the phrase *eliminating the vertex*.

The nested dissection heuristic divides the graph of the model problem into five subsets: a cross dividing the grid into four equal parts, and the four $(n-1)/2 \times (n-1)/2$ subgrids. See Figure 2. By numbering (eliminating) the cross vertices last the vertices in the subgrids can be eliminated without edges forming between vertices in different subgrids. The vertices of the subgrids are recursively ordered by the same rule.

If the inital cross of vertices is considered to be at level l there will be four crosses at level $l-1$ and 4^{l-k} crosses at level k. For the model problem $l = \log_2(n+1)$. All vertices at a given level are eliminated before those at the

next higher level. With this variable ordering the matrix A has the form:

$$A = \begin{pmatrix} Q_1 & & & & C_1^t \\ & Q_2 & & & C_2^t \\ & & Q_3 & & C_3^t \\ & & & Q_4 & C_4^t \\ C_1 & C_2 & C_3 & C_4 & S_l \end{pmatrix}.$$

Q_i is an $(N-1)/4 \times (N-1)/4$ matrix corresponding to the vertices in quadrant i. S_l is a $(2n-1) \times (2n-1)$ matrix corresponding to the cross of vertices separating the four subgrids. C_i is a $(2n-1) \times (N-1)/4$ matrix corresponding to the edges between subgrid i and the cross. Each Q_i in turn is a matrix with the same structure as A.

The LDL^t factorization of A for the model problem using the nested dissection ordering is:

$$\begin{pmatrix} Q_1 & & & & C_1^t \\ & Q_2 & & & C_2^t \\ & & Q_3 & & C_3^t \\ & & & Q_4 & C_4^t \\ C_1 & C_2 & C_3 & C_4 & S_l \end{pmatrix} =$$

$$\begin{pmatrix} L_1 & & & & \\ & L_2 & & & \\ & & L_3 & & \\ & & & L_4 & \\ \tilde{C}_1 & \tilde{C}_2 & \tilde{C}_3 & \tilde{C}_4 & L_s \end{pmatrix} \begin{pmatrix} D_1 & & & & \\ & D_2 & & & \\ & & D_3 & & \\ & & & D_4 & \\ & & & & D_s \end{pmatrix} \begin{pmatrix} L_1^t & & & & \tilde{C}_1^t \\ & L_2^t & & & \tilde{C}_2^t \\ & & L_3^t & & \tilde{C}_3^t \\ & & & L_4^t & \tilde{C}_4^t \\ & & & & L_s^t \end{pmatrix}$$

where L_i is a lower triangular matrix with the same general structure as L.

The k^{th} level vertices reside in 4^{l-k} crosses, each enclosed in a box composed of vertices at higher levels. Each cross has $2^{(k+1)} - 3$ vertices, and each spoke of the cross has $2^{(k-1)} - 1$ vertices. To order the vertices in a cross number the vertices in the right horizontal spoke from right to left, continue numbering in the left spoke from right to left, and finish numbering in the vertical separator from the top to the bottom. See Figure 3. To eliminate a cross:

a) Eliminate the left and right horizontal spokes of the cross. These are independent groups of vertices.

b) Eliminate the remaining vertical separator.

The elimination of the crosses at the k^{th} level are all independent. We will refer to these as k^{th} level subproblems.

The elimination of the vertices at lower levels couples each of the horizontal spoke vertices to all of the vertices in the box surrounding it, which is made up of the vertical separator and half of the vertices in the box surrounding the entire cross, and to no vertices outside of the box. After the elimination of the

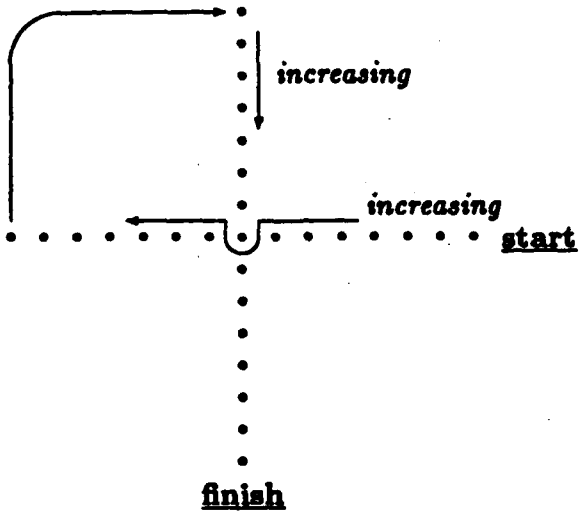


Figure 3: Vertex numbering within a cross.

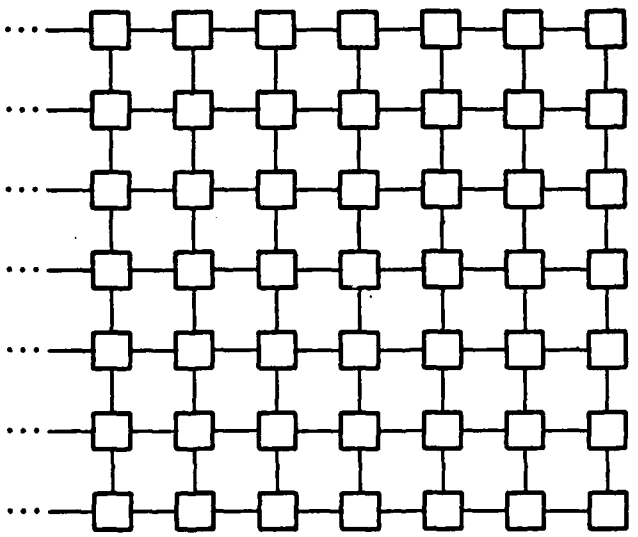


Figure 4: 7 by 7 processor array.

horizontal spokes the vertical separator vertices are coupled to all of the vertices in the box surrounding it. Let p be the number of vertices in a spoke. Let q be the number of vertices in the surrounding box. Then the elimination of a spoke of vertices from a k^{th} level cross corresponds to a partial factorization of a dense matrix F:

$$(3.1)\quad F \equiv \begin{pmatrix} S & C^t \\ C & B \end{pmatrix} = \begin{pmatrix} L & 0 \\ CL^{-t}D^{-1} & I \end{pmatrix} \begin{pmatrix} D & 0 \\ 0 & \tilde{B} \end{pmatrix} \begin{pmatrix} L^t & D^{-1}L^{-1}C^t \\ 0 & I \end{pmatrix}.$$

S is a $p \times p$ symmetric positive definite matrix, and $\tilde{B} = B - CS^{-1}C^t$ is the $q \times q$ matrix B modified by the elimination of the spoke. The matrix F is a selection of rows and columns from the matrix A_τ (for $\tau = 4^l[1 - 2^{-k+2}(1 - 2^{-k})]$) from which zero elements have been removed.

Using the same notation as in section 2 the factorization proceeds by calculating

$$\begin{pmatrix} L_i & \\ C_i & I \end{pmatrix} F^{(i)} \begin{pmatrix} L_i^t & C_i^t \\ & I \end{pmatrix}$$

from

$$\begin{pmatrix} L_{i-1} & \\ C_{i-1} & I \end{pmatrix} F^{(i-1)} \begin{pmatrix} L_{i-1}^t & C_{i-1}^t \\ & I \end{pmatrix}$$

where

$$F^{(i-1)} = \begin{pmatrix} D_{i-1} & & \\ & d_i & \bar{v}_i^t \\ & \bar{v}_i & \tilde{F}_i \end{pmatrix}, \qquad F^{(i)} = \begin{pmatrix} D_{i-1} & & \\ & d_i & \bar{0}^t \\ & \bar{0} & F_i \end{pmatrix},$$

and $F_i = \tilde{F}_i - \bar{v}_i\bar{v}_i^t/d_i$. Let $f_{kj}^{(i)}$ be the (k,j) element of $F^{(i)}$ and $f_{kj}^{(0)}$ be the (k,j) element of F. So for each column j in $F^{(i-1)}$, $j > i$, we calculate

$$f_{kj}^{(i)} = f_{kj}^{(i-1)} - \frac{(\bar{v}_i)_{k-i}}{d_i}(\bar{v}_i)_{j-i} = f_{kj}^{(i-1)} - \frac{f_{ji}^{(i-1)}}{f_{ii}^{(i-1)}} f_{ki}^{(i-1)} \quad \text{for } k \geq j$$

since $F^{(i-1)}$ is symmetric.

The partial factorization stops at

$$\begin{pmatrix} L_p & \\ C_p & I \end{pmatrix} F^{(p)} \begin{pmatrix} L_p^t & C_p^t \\ & I \end{pmatrix}.$$

The vectors $\{\bar{v}_i\}$ are identical to the vectors $\{\bar{u}_j\}$ described in section 2, for some j, with the zero elements removed. The elimination of the vertical separator is described similarly.

4. The Architecture and Its Uses. The machine is an $n \times n$ grid of processors. Each processor can communicate with its four nearest neighbors. Processors on one side of the grid have access to an external communication network. See Figure 4. This is an MIMD architecture, and each processor has both instruction and data memory, and can do floating point addition, subtraction, multiplication, and division. The data memory can hold up to $3.5n + O(1)$ floating point numbers, while the program memory can hold a program whose size is independent of n .

The memory requirements are necessary for our algorithm. The program consists of steps of m ($m < 4$) simultaneous writes, followed by up to $4 - m$ simultaneous reads, followed by some arithmetic. The writes and reads are not on the same lines, and the written values can remain on the data path throughout the step to insure a correct read by the destination processor. This architecture scales since the length of the interprocessor communication lines remains fixed as n increases. There is neither a global instruction path, nor a global clock, therefore all wires external to the processors are short. But the memory requirements do grow linearly in n for our algorithm.

The nested dissection algorithm described in section 3 calculates the symmetric factorization of A by partially factoring a sequence of small dense matrices. Systolic algorithms perform very well on dense problems of these sorts [9], and these algorithms can be run within the topology of our processor array. Again let p be the number of vertices in a spoke or separator to be eliminated. Let q be the number of vertices in the surrounding box. We treat the partial factorization of equation (3.1) as two operations. First is a *reduction* phase which calculates the first p columns of the lower triangular factor ($\{\bar{v}_i/d_i\}$), and the first p columns of $F^{(p)}$ ($\{d_i\}$). The second is an *update* which produces the last q columns of $F^{(p)}$.

A systolic architecture sufficient to perform the reduction phase is a $p(p+1)/2$ element triangular processor array. See Figure 5. We will refer to processors by their (column, row) coordinates. Assume that processor $(i, 1)$ has access to the matrix elements (on and below the diagonal) of column i. By sending the elements of column 1 from processor $(1, 1)$ to the left through the processors in row 1, we can calculate

$$f_{kj}^{(1)} = f_{kj}^{(0)} - \frac{(\bar{v}_1)_{k-1}(\bar{v}_1)_{j-1}}{d_1}$$

in processor $(j, 1)$ for all $k \in \{j, p+q\}$. If the elements $f_{kj}^{(1)}$ are sent to processor $(j, 2)$ as soon as they are available we can begin to calculate

$$f_{kj}^{(2)} = f_{kj}^{(1)} - \frac{(\bar{v}_2)_{k-2}(\bar{v}_2)_{j-2}}{d_2}$$

immediately. This works since processor $(2, 1)$ is producing its results first, receiving the data from processor $(1, 1)$ directly, and its results are the d_2 and $\bar{v}_2$

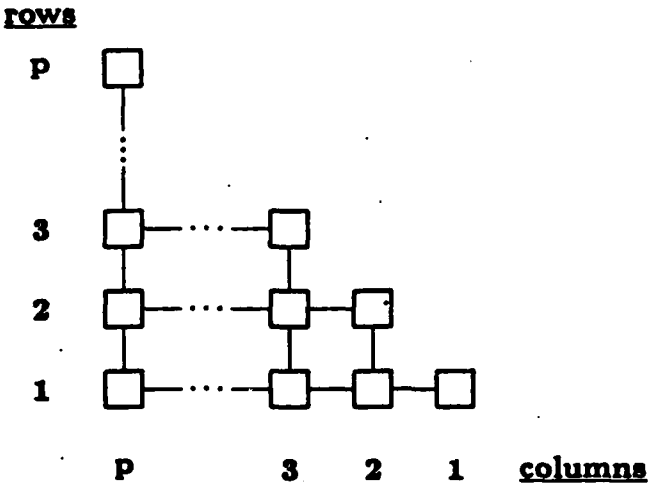


Figure 5: Processor array for the reduction algorithm.

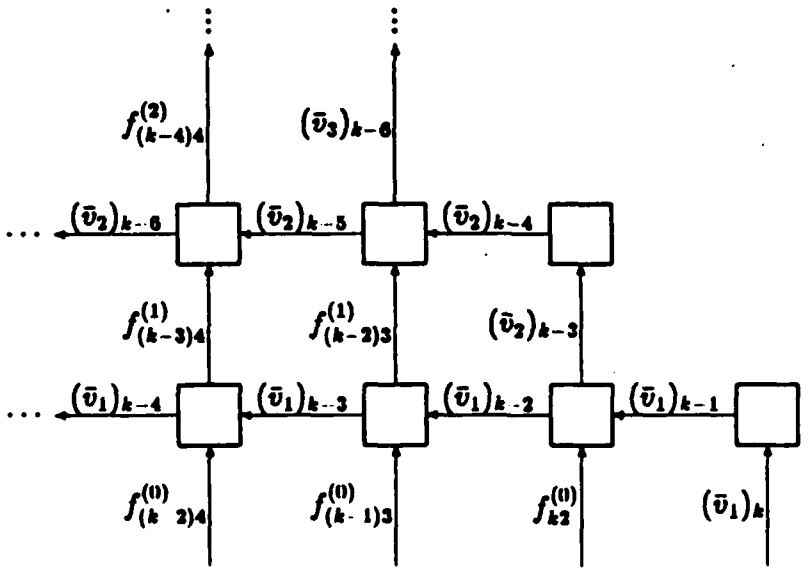


Figure 6: Snapshot of reduction algorithm.

the three other directions at the beginning of the next step. Each processor uses the data it receives if appropriate. Processors which already have their solutions ignore all incoming data, so the wave front does not escape the square. The horizontal spokes do not let information pass across them vertically, so the x_p wave front stops at the horizontal spokes except for a pulse that continues up the vertical separator. As soon as x_p passes though processor $p-1$ it has its solution x_{p-1}. The step after x_p has been sent on its way processor $p-1$ sends out x_{p-1}, starting its own wave front. The two wave fronts do not conflict, one being properly nested inside the other. Two steps later processor $p-2$ will start its own wave front with the x_{p-2} datum. See Figure 13 for a pictorial description of this process. This continues until the center of the cross is ready to broadcast its value. The horizontal spokes will send the information both upwards and downwards, as per the convention, and the broadcast will cover the entire square. But the next wave front is again trapped by the horizontal spokes, this time in the upper half of the square, as are all of the rest of the wave fronts in step (a).

Let one backward solve step consist of either 1 or 3 writes, followed by 1 read, followed by 1 multiplication and 1 subtraction.

Lemma 6.1. *Part (a) for a level k backward solve subproblem takes* $3 \cdot 2^k - 6$ *backward solve steps.*

Proof: The vertical separator contains $2^k - 1$ processors. It takes $2^k - 2$ steps for the first information to arrive at the top of the separator. It takes another $2^k - 3$ steps for the rest of the vertical separator information to arrive. This top processor then waits one step before broadcasting its value, which takes $2^k - 2$ steps to cover the top half of the square. ∎

Part (b) is practically the same algorithm except that the vertical–horizontal conventions of before are reversed. If data from a wave front enters a processor from the right or left it sends it on to the left or right respectively at the next step, but if it enters from the top or bottom the processor sends it on in the other three directions. The horizontal spokes start with the leftmost processors. The right and left spokes are backward solved concurrently since nothing passes across the now inactive vertical separator processors.

Lemma 6.2. *Part (b) of a level k backward solve subproblem takes* $2^{k+1} - 6$ *backward solve steps.*

Proof: The horizontal spokes each contain $2^{k-1} - 1$ processors. It takes $2^{k-1} - 2$ steps for the rightmost processor of each spoke to see its first information. $2^{k-1} - 3$ steps later it has seen the rest of the wave fronts arrive. After waiting one step this last processor broadcasts its value, which takes $2^k - 2$ steps to cover its half of the square. ∎

The subproblems on a given level can all be solved in parallel. Level l is not a special case for the backsolve, and level 1 is trivial as there is nothing to do.

Theorem 6.1. *The forward solve algorithm takes* $22n + (6r - 26)\log_2(n+1) - 4r - 24$ *forward solve steps and* $17.5n + (9r - 24)\log_2(n+1) - 15r - 18$ *communication steps.*

Proof: This result follows from an analysis analogous to the factorization operation count, using similar level 1 and level l improvements. The deliberate transpose of data in step b)vii) of the factorization is not needed, nor is any other such data movement. ∎

If the forward solve is calculated during the factorization it does not need the row entries of L. This reduces the data storage requirements from $3.5n$ floating point numbers to $2n$. The separate forward solve algorithm can be modified to take advantage of the same storage savings, but it significantly increases the data movement required and involves a more complex program. Notice that $r-1$ additional right hand sides only increase the work by $O(r\log_2(n+1))$ steps, yet there is an overhead of $O(n)$ to get started.

For now consider solving for a single right hand side. The vector $\bar{z}$, $\bar{z} = L^{-1}\bar{b}$, has been calculated and z_j stored in processor j. If not calculated during the factorization we next calculate the diagonal solve, $y_j = z_j/d_j$, in all the processors simultaneously. Finally we calculate the backward solve,

$$x_j = y_j - \sum_{i=j+1}^{N} \frac{1}{d_j}(\bar{u}_j)_{i-j}x_i.$$

Once an x_i is calculated it is broadcast to all of the processors containing a y_j, $j < i$, such that $(\bar{u}_j/d_j)_{i-j} \neq 0$. Again we have subproblems arising from the variable ordering. Within a square grid bordered by processors which already contain their associated solution, x_i, we calculate the solutions associated with a cross of processors. The results are sent to processors in the four subgrids of the square. When they have all been received crosses in these subgrids can be backward solved in parallel and the results in turn broadcast to their subgrids. We overlap the backward solve of the cross and the broadcast of the results, and divide this process into two parts:

a) Backward solve and broadcast the vertical separator.
b) Backward solve and broadcast the left and right horizontal spokes simultaneously.

At the start of part (a) the processor at the bottom of the vertical separator (call this vertex p) already has its solution x_p, and sends this result to its three neighboring processors inside the square. At the next step these processors pass x_p to their neighbors who haven't already received it. The information continues to move out in a wave front through the grid. If the information is received from above or below, the receiving processor continues sending it in the original direction. If it is received from the left or the right the processor sends it out in

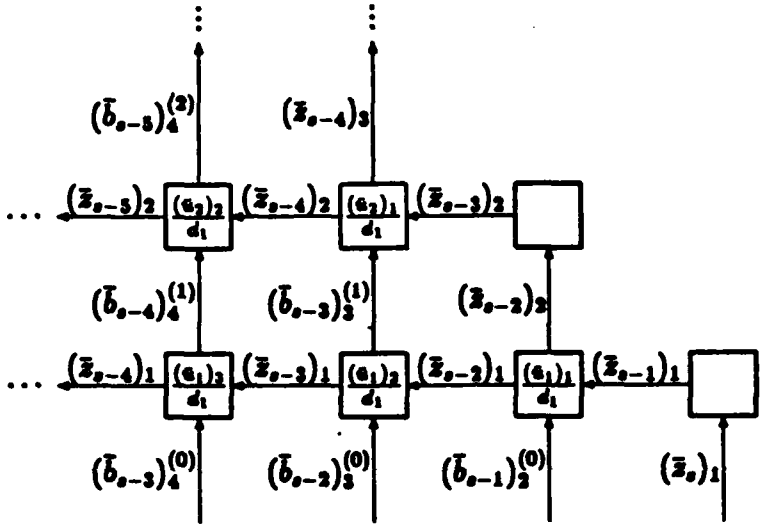


Figure 11: Snapshot of forward solve reduction algorithm.

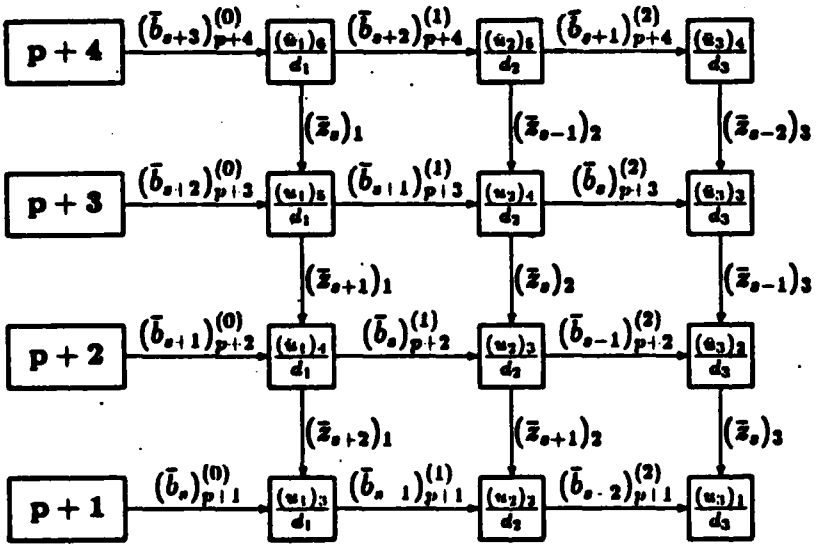


Figure 12: Snapshot of forward solve update algorithm.

We again solve dense subproblems, the forward solve of a k^{th} level cross of vertices, using systolic algorithms. In the factorization a spoke of vertices is reduced using a triangular array of processors. Number these vertices as ν to $\nu+p-1$. We forward solve for $\{(\bar{z}_s)_j | j \in \{\nu,\ldots,\nu+p-1\}\}$ for all $s \in \{1,\ldots,r\}$ using the same processor array. Each spoke processor j loads the idle processors above it with

$$\{(\bar{u}_i/d_i)_{j-i} | i \in \{\nu,\ldots,\nu-1+j\}\},$$

with $(\bar{u}_i/d_i)_{j-i}$ in row $i-\nu$. Processor ν sends the values $\{(\bar{z}_s)_\nu\}$ through row 1, modifying the values $\{(\bar{b}_s)_j\}$ from the other processors. This finishes the calculation for

$$\{(\bar{z}_s)_{\nu+1},\ s=1,\ldots,r\},$$

each element of which is sent down row 2 as soon as it is calculated to continue the reduction. Similarly, $\{(\bar{z}_s)_{\nu+i}\}$is sent through row $i-1$, finishing the calculation of $\{(\bar{z}_s)_{\nu+i+1}\}$. As in the factorization the calculation is also performed in a triangular processor array below the horizontal spoke. This process takes $p-1$ communication steps* to load the processor array, and $2p+r$ forward solve steps consisting of 2 writes, 2 reads, 1 multiplication, and 1 subtraction. See Figure 11 for a snapshot of the algorithm with $\nu=1$.

In the factorization we update the column entries of a box of vertices one side at a time. The forward solve updates the elements of the right hand sides corresponding to these same vertices using the same rectangular processor array. For example, consider the update of the left side using the results of a spoke forward solve,

$$\{(\bar{z}_s)_j \mid s=1,\ldots,r;\ j=\nu,\ldots,\nu-1+p\},$$

stored at the top. Each processor j in the left side fills its corresponding row with the elements

$$\{(\bar{u}_i/d_i)_{j-i} \mid i \in \{\nu,\ldots,\nu-1+p\}\},$$

placing $(\bar{u}_i/d_i)_{j-i}$ in the processor directly below where $\{(\bar{z}_s)_i\}_{s=1}^r$ is stored. The data movement is the same as in the factorisation, with the entries to be modified, $\{(\bar{b}_s)_j\}$, moving from the left to the right, and the information doing the modifying, $\{(\bar{z}_s)_i\}$, moving from the top to the bottom. This takes p communication steps to load $\{(\bar{u}_i/d_i)\}$, $p+m+r-2$ forward solve steps (m being the number of vertices from the left side being updated), and $p+r-1$ communication steps to return the results. See Figure 12 for a snapshot of the algorithm with $\nu=1$, $p=3$, and $m=4$.

---

* This assumes that writing the same data in two different directions costs the same as writing it only once during a communication step. Otherwise $2p-2$ communication steps are required.

factorisation, except that some efficiencies can be exploited in levels 1 and l. In both of these the given algorithm will work, but in level l there is no surrounding box of active processors for the single subproblem, nor other subproblems with which it must synchronize, so it can simply ignore the parts of the algorithm which are meaningless for it. This takes $15 \cdot 2^{l-1} + 3$ factorization steps and $8 \cdot 2^{l-1} - 1$ communication steps for step (a), and $6 \cdot 2^{l-1} + 3$ factorization steps and $10 \cdot 2^{l-1} + 1$ communication steps for step (b). The level 1 subproblems can also be done more efficiently, requiring a complexity of 41 factorization steps. See [11] for more details. This leads us to:

Theorem 5.1. *The parallel factorization of an $N \times N$ symmetric positive definite matrix A whose matrix graph is isomorphic to a $\sqrt{N} \times \sqrt{N}$ finite element mesh takes $63n - 24\log_2 n - 115$ factorization steps and $46n - 19\log_2 n - 128$ communication steps, if $n \equiv \sqrt{N} = 2^l - 1$.*

6. The Parallel Triangular Matrix Solves. Having factored A into LDL^t we still must solve three problems,

$$\begin{aligned} L\bar{z} &= \bar{b} && \text{forward solve} \\ D\bar{y} &= \bar{z} && \text{diagonal solve} \\ L^t\bar{x} &= \bar{y} && \text{backward solve,} \end{aligned}$$

to solve the system $A\bar{x} = \bar{b}$. After the factorization row j of L, D, and L^t are in processor j. We assume that b_j has also been loaded into processor j.

The forward solve can be calculated in conjunction with the factorisation by adding $(\bar{b}^t\ 1)$ as an extra row, and its transpose as an extra column. In general, if we had r right hand sides, $B = [\bar{b}_1 \dots \bar{b}_r]$, the algorithm produces

$$\begin{pmatrix} A & B^t \\ B & I \end{pmatrix} = \begin{pmatrix} L & 0 \\ BL^{-t}D^{-1} & I \end{pmatrix} \begin{pmatrix} D & 0 \\ 0 & \tilde{I} \end{pmatrix} \begin{pmatrix} L^t & D^{-1}L^{-1}B^t \\ 0 & I \end{pmatrix},$$

where $\tilde{I} = I - BA^{-1}B^t$ is never calculated. The diagonal solve is provided here also. For r right hand sides the cost is

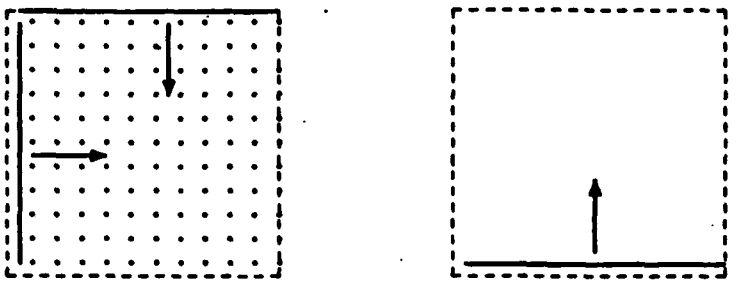
$$r(10\log_2(n+1) - 2) \quad \text{factorization steps}$$

and

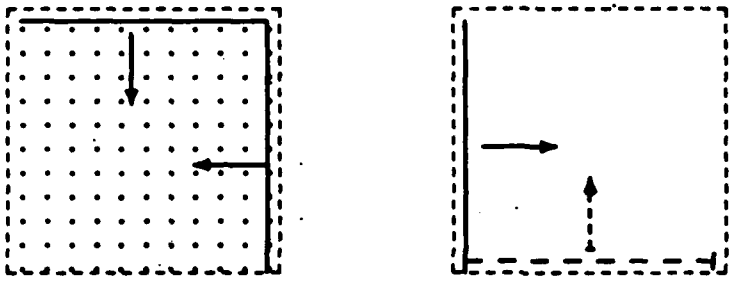
$$r(9\log_2(n+1) - 5) \quad \text{communication steps.}$$

The vectors $\{\bar{b}_s\}$ are not always be available at the time of factorization. For completeness sake we describe a forward solve algorithm, which is based on mimicking the factorization algorithm. The calculation is

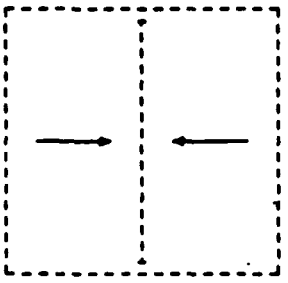
$$(\bar{z}_s)_j = (\bar{b}_s)_j - \sum_{i=1}^{j-1} \frac{1}{d_i}(\bar{u}_i)_{j-i}(\bar{z}_s)_i .$$



v) Update the top. Return.

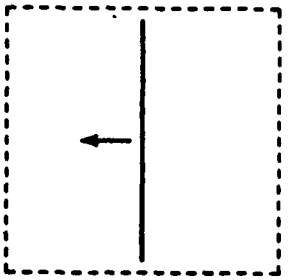


vi) Update the right. Return (and load data for (vii)).

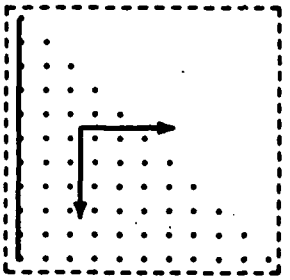


vii) Transpose the update information.

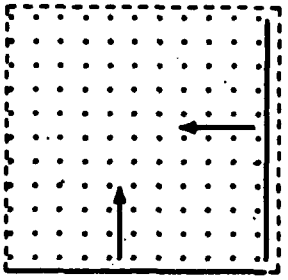
Figure 10: Phases of step (b) of the factorization. Information flow is indicated by the arrows. Data sources are represented by the heavy solid lines. Stippled areas indicate processors engaged in arithmetic.



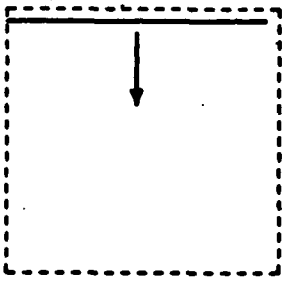
i) Move the separator data.



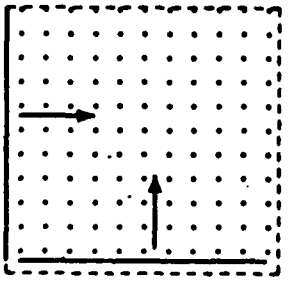
ii) Reduce the separator.



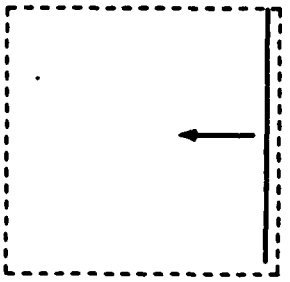
iii) Update the bottom.



Return.



iv) Update the left.



Return.

require simultaneous access to shared processors. This organization also allows step (a) of all the subproblems in level k to be calculated concurrently. Not all subproblems are of the form described. Those crosses on the edge of the grid are missing one or more sides of the surrounding box. But the crucial processors, the idle ones in the four quadrants of the cross, are always there. Processors solving border subproblems wait at times to synchronize with the calculations of the interior subproblems.

Step (b), the elimination of the vertical separator, is very similar to step (a). Refer to Figure 10 for a pictorial description and below for a description of the differences from step (a).

i) Send the column data in the vertical separator processors to the left. Store the data in the idle processors just inside the left side of the surrounding square. This is done so that the reduction algorithm can use all of the idle processors inside the square.

ii) Reduce the vertical separator. The results are stored in two places, in the idle processors next to the right side of the surrounding square and in the idle processors next to the bottom side of the square.

iii) Update the vertices on the bottom. The update information, moving from right to left, is saved in the idle processors next to the left side of the surrounding box of active processors.

iv) Update the vertices on the left. The update information, moving from the bottom to the top, is saved in the idle processors next to the top of the box.

v) Update the vertices on the top.

vi) Update the vertices on the right.

vii) Transpose the update information. We still need to store the values

$$\{\frac{1}{d_j}(\bar{v}_j)_{k-j} | j \in \text{the vertical separator}, \ k > j\}$$

for each processor k in the vertical separator. First send the information up into the interior processors from the bottom. Next the processors to the right of the separator send these values left into the vertical separator, followed by the processors on the left sending their values right into the vertical separator.

Lemma 5.2. *Step (b) at level k takes $62 \cdot 2^{k-1} - 11$ factorization steps and $51 \cdot 2^{k-1} - 11$ communication steps.*

This organization allows all of the level k subproblems to perform step (b) in parallel. The border subproblems again represent a straightforward reduction of the given algorithm, with processors waiting when appropriate. This gives all the information required to calculate the parallel complexity of the parallel

iii) Update the vertices on the right. Include the top right vertex of the surrounding rectangle, but not the bottom right one. The rectangular processor array includes the active processors on the top and the right. The update information flows from the bottom up, the column entries to be updated move from the right to the left, and the results end up in the far left column of idle processors, plus one active processor at the top. The results (and $\{(\bar{v}_j/d_j)_{k-j}\}$) are then sent back to the appropriate processors.

iv) Update the vertices on the bottom. Include the bottom right vertex, but not the bottom left. The rectangular processor array consists of the active processors on the bottom and the bottom half of the active processors on the right, and the idle processors below the horizontal spoke. The update information flows from the left to the right, the column entries to be updated flow from the bottom up, and the results end up in the processors just below the horizontal spoke processors. The results (and $\{(\bar{v}_j/d_j)_{k-j}\}$) are then sent back to the appropriate processors. Simultaneously the update information held in the processors on the left side above the horizontal spoke is sent to the right side. The values

$$\{\frac{1}{d_j}(\bar{v}_j)_{k-j} | j \in \text{the horizontal spoke}, \ k > j\}$$

are also sent to each processor k in the horizontal spoke while the update proceeds in the processors below the spoke.

v) Update the vertices on the top. Include the top left vertex, but not the top right one. The rectangular processor array includes the active processors on the top and the top half of the active processors on the left. The update information flows from the right to the left, the column entries to be updated flow from the top to the bottom, and the results end up in the processors just above the horizontal spoke processors. The results (and $\{(\bar{v}_j/d_j)_{k-j}\}$) are then sent back to the appropriate processors.

Define a factorization step to be two simultaneous writes, followed by two simultaneous reads, followed by either one division or two multiplications and one addition. Define a communication step to be one write followed by one read.

Lemma 5.1. *Step (a) of the factorization at level k takes $43 \cdot 2^{k-1} - 13$ factorization steps and $28 \cdot 2^{k-1} - 8$ communication steps.*

Proof. A simple counting argument and the exact specification of the reduction and update algorithms are all that is needed. See [11] for details. ∎

This particular organization of step (a) allows both the right and left horizontal spokes to be eliminated simultaneously because the two eliminations never

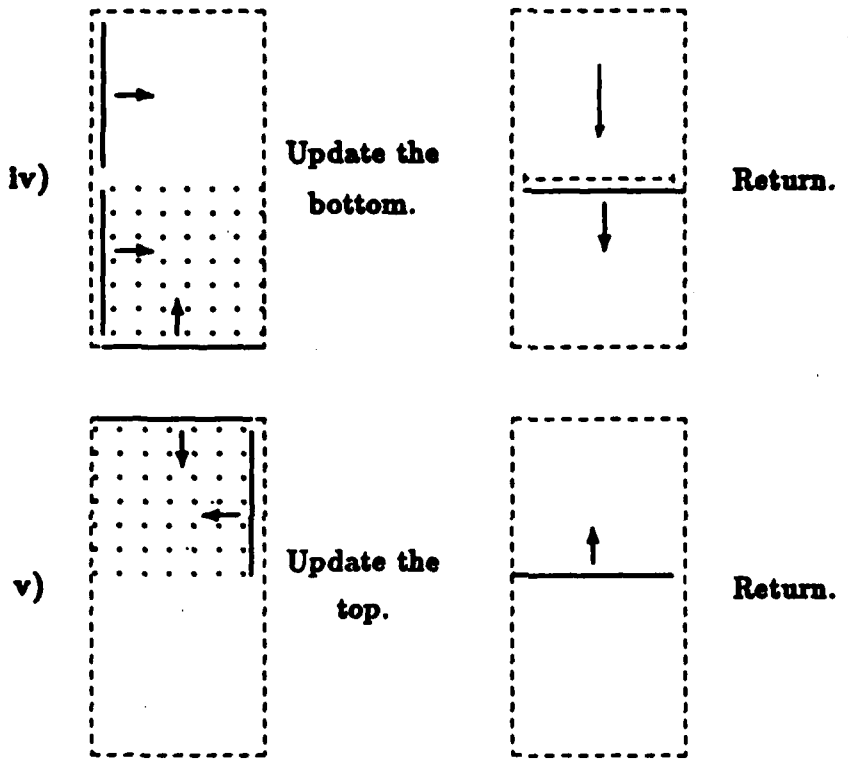
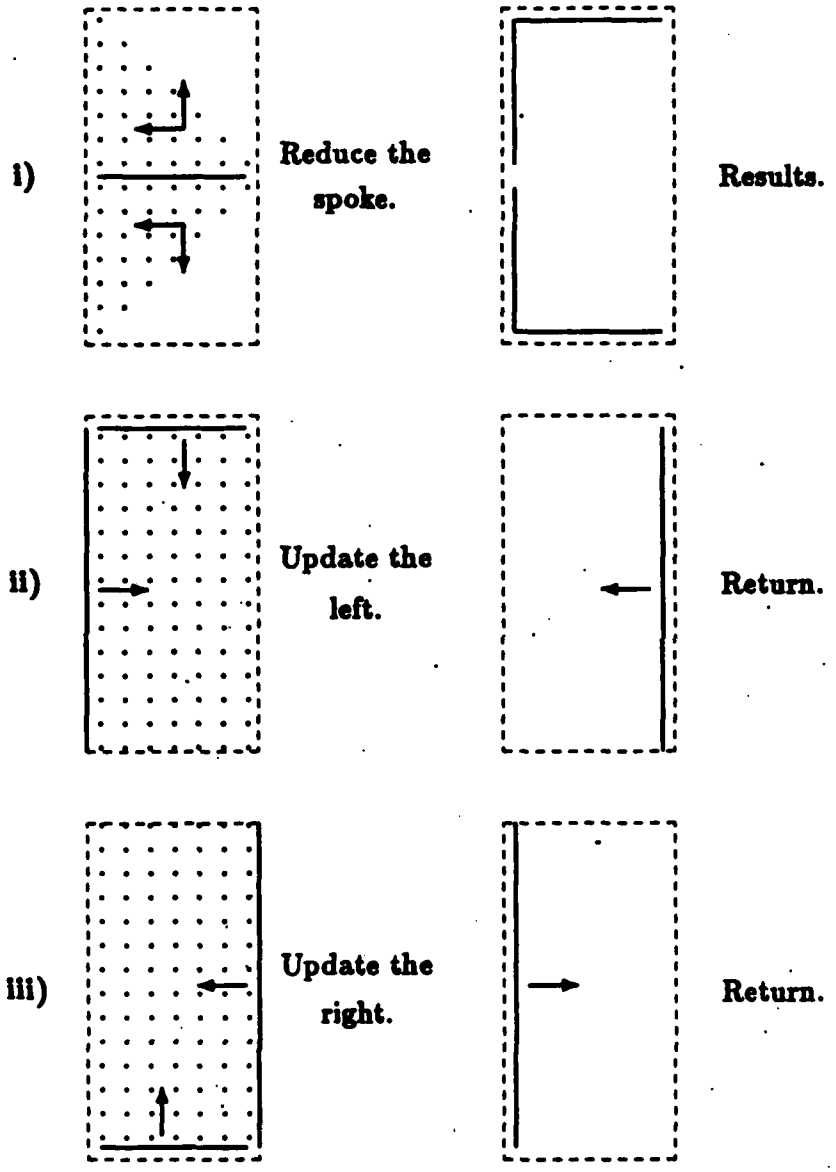


Figure 9: Phases of step (a) of the factorization. Information flow is indicated by the arrows. Data sources are represented by the heavy solid lines. Stippled areas indicate processors engaged in arithmetic.


i)
Reduce the spoke.
Results.
ii)
Update the left.
Return.
iii)
Update the right.
Return.

complete, with results residing in the processors in column 1. Figure 8 shows a snapshot of the architecture near the beginning of the algorithm for a particular ordering (and with $p = 3$ and $m = 4$). Again, this algorithm will be described in more detail in [11].

5. The Parallel Factorization. We use a parallel processor whose interconnection topology approximates the graph of the matrix A. Each vertex i is represented by a processor, which initially contains the nonzero matrix elements (on and below the diagonal) of column i. After the i^{th} vertex is eliminated the processor will contain the nonzeroes in the i^{th} rows of L, D, and L^t. Once the vertex is eliminated the processor is free to be used in the elimination of other vertices, and we call it *idle*. Until then the processor is called *active*. We use these idle processors to implement the systolic algorithms introduced in the previous section. Henceforth we will use the terms vertex and processor interchangeably, using whichever is most appropriate. We will also refer to the reduction or update phases in a spoke elimination as reducing or updating the vertices associated with the values being calculated.

Step (a) of the cross elimination for a subproblem at level k is pictorially represented in Figure 9, and described below.

i) Reduce the horizontal spoke. The reduction is done simultaneously in idle processors above and below the spoke processors. The results $\{(d_j, \bar{v}_j/d_j)\}$ are saved in four different groups of processors, in the left edges of the two triangular processor arrays and in the idle processors next to the top and the bottom of the box defined by the surrounding active processors.

ii) Update the vertices on the left. Include the bottom left vertex of the surrounding rectangle, but not the top left one. The rectangular processor array to calculate the update consists of the active processors on the left and the bottom, and all of the interior idle processors. The update information $\{(d_j, \bar{v}_j/d_j)\}$ flows down from the top, the column entries to be updated move from left to right, and the results end up in the far right column of idle processors (including one active processor at the bottom). The update information is saved in the horizontal spoke processors as it moves back through them. Next we need to return the results of the update to their original processors. We also need to save the values

$$\{(\frac{1}{d_j}\bar{v}_j)_{k-j} | j \in \text{ the horizontal spoke}, \ k > j\}$$

at processor k. These elements represent part of the k^{th} row of L. Fortunately the update algorithm left these elements in the processors between processor k and the processor holding k's updated column elements. When the results are sent back, the set of values $\{(\bar{v}_j/d_j)_{k-j}\}$ can be sent to processor k as well.

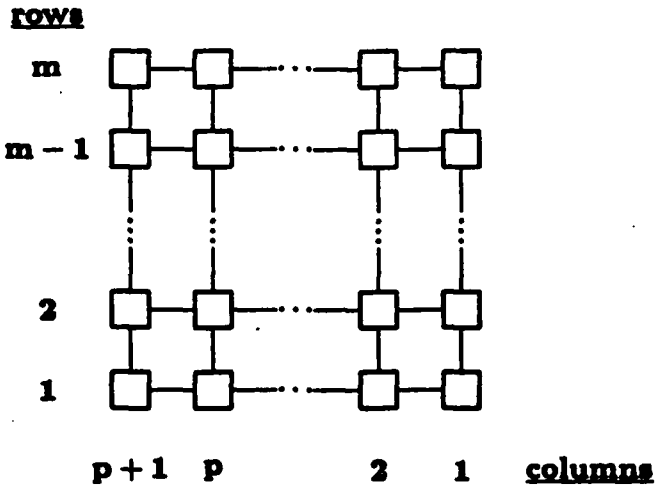


Figure 7: Processor array for the update algorithm.

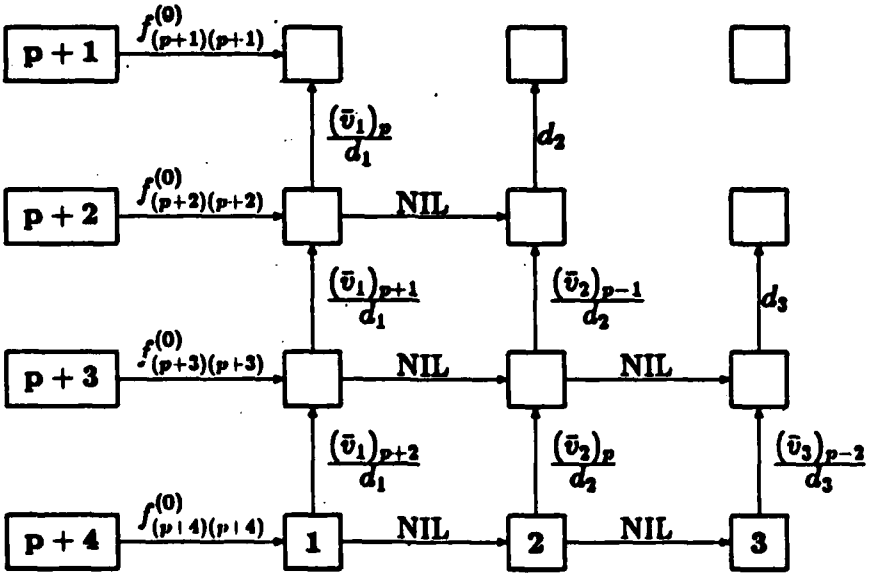


Figure 8: Snapshot of update algorithm. Numbers in processors indicate column data stored there.

elements needed to calculate $f_{kj}^{(2)}$. This continues with processor $(m, m-1)$ calculating d_m and $\bar{v}_m$, while row m calculates $f_{kj}^{(m)}$ for $j > m$. In $3p+q$ steps the reduction phase is completed. More detail on this algorithm will be available in a Stanford technical report [11]. See Figure 6 for a snapshot of the algorithm. This systolic algorithm incorporates no new ideas [8][9], but it is what is needed to calculate the reduction phase efficiently on our machine.

The update phase requires access to the vectors $\{(d_i, \bar{v}_i/d_i) | i = 1, \ldots, p\}$ which are calculated in the reduction phase. Call this the *update information*. Since the factorization preserves symmetry, only the elements on and below the diagonal need to be updated. So for column j of the matrix only the elements d_i and $(\bar{v}_i/d_i)_{k-i}$, $k \geq j$ are used.

An architecture to update a subset of size m of the q columns is an $m \times (p+1)$ rectangular array of processors. See Figure 7. Assume each processor in row 1, except processor $(p+1, 1)$, has access to $(d_i, \bar{v}_i/d_i)$ for some distinct $i \in \{1, \ldots, p\}$. Assume each processor in column $p+1$ has access to the elements on and below the diagonal of a column j of the matrix for some $j \in \{p+1, \ldots, p+q\}$.

The algorithm begins with processor $(p+1, 1)$ sending its column entries to the right through the processors in row 1. Since the matrix column assigned to processor $(p+1, 1)$ is arbitrary, the row 1 processors assume that it is column $p+1$ and expect to calculate

$$f_{j,p+1}^{(p)} = f_{j,p+1} - \sum_{i=1}^{p} \frac{(\bar{v}_i)_{p+1-i}(\bar{v}_i^t)_{j-i}}{d_i}$$

as $f_{j,p+1}$, $j \in \{p+1, \ldots, p+q\}$, moves through row 1. If instead processor $(p+1, 1)$ contains matrix column s, then for the first $s-p-1$ steps it sends a NIL value to tell the row 1 processors that nothing should be done during these steps. As soon as a real number is sent the processors can recognize this and start calculating

$$f_{js}^{(p)} = f_{js} - \sum_{i=1}^{p} \frac{(\bar{v}_i)_{s-i}(\bar{v}_i)_{j-i}}{d_i}$$

for successive $j \in \{s, \ldots, p+q\}$. Processor $(r, 1)$, $r \in \{1, \ldots, p\}$, must wait $p-r$ steps for its first value from processor $(p+1, 1)$ and remains delayed that much throughout the algorithm. As soon as the j^{th} value from processor $(p+1, 1)$ reaches processor $(r, 1)$, $(\bar{v}_r/d_r)_{j+p-r}$ is no longer needed (with the exceptions that d_r and $(\bar{v}_r/d_r)_{s-r}$ are saved locally) and can be sent to processor $(r, 2)$. Thus if processor $(p+1, 2)$ starts sending its NILs and column entries through row 2 one step after processor $(p+1, 1)$ starts, row 2 can update the matrix column associated with processor $(p+1, 2)$. The same analysis holds true for the other rows of the processor array, where each row begins one step later than the previous row. In $p+m+q$ steps the update of these m columns of the matrix is

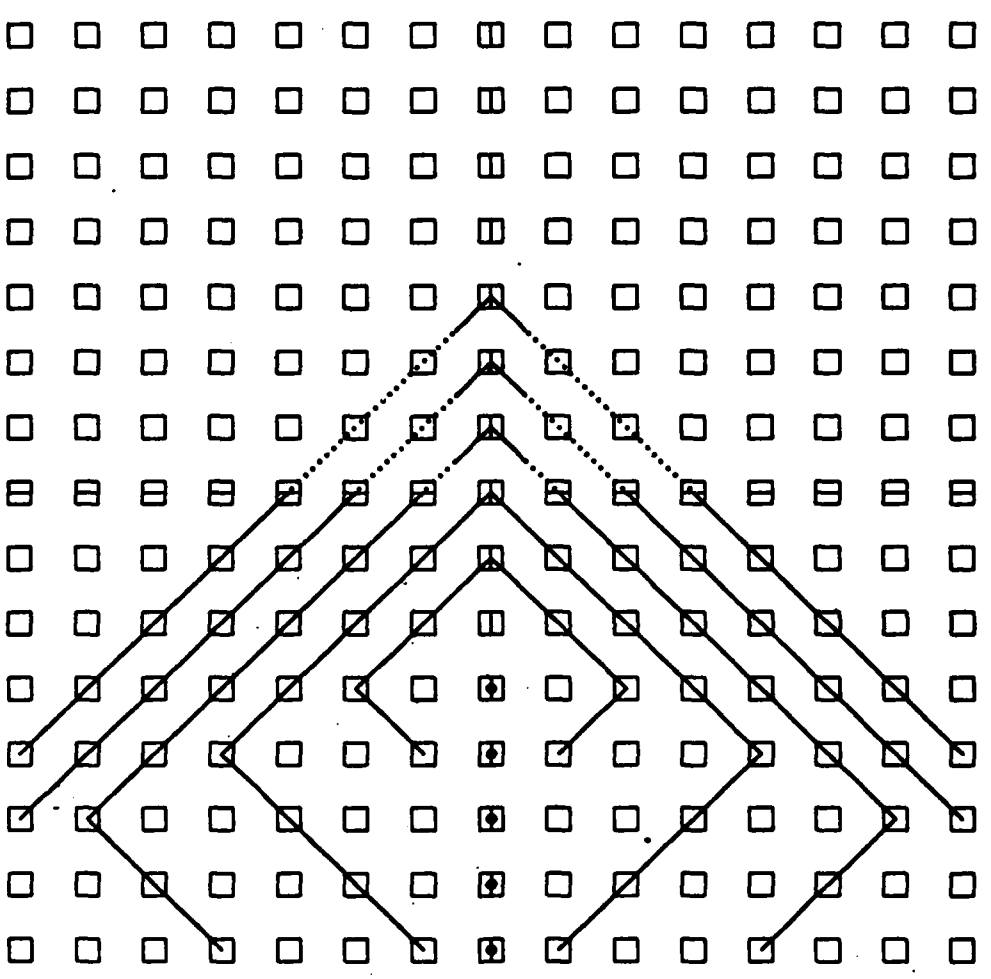


Figure 13: Snapshot of wavefronts developing in the backsolve algorithm. Circles indicate the processors which have spawned wavefronts. The vertical and horizontal spoke processors are marked with vertical and horizontal solid lines.

So we have

Theorem 6.2. *The backward solve for the model problem takes* $10 \cdot n - 12 \log_2(n+1) - 8$ *backward solve steps.*

The backward solve algorithm on a spoke is similar to the Kung and Leiserson linear systolic array algorithm [9]. This is then embedded into our broadcast algorithm.

The backward solve algorithm wastes some time broadcasting results to processors where they are not needed. The simplicity of the program for the algorithm outweighs the small amount of savings which are available. A more serious drawback is the lack of savings for multiple right hand sides. The complete algorithm must be rerun for each $\bar{y}_s$. This follows from the backward solve algorithm being based on a linear systolic array algorithm.

Gannon describes a different backward solve algorithm which is suitable for multiple right hand sides. His approach uses a two dimensional systolic array algorithm and only broadcasts the results when and where they are needed. The backward solve of a cross is again a subproblem. Preliminary to this the right hand sides corresponding to these vertices are updated by the results available in the surrounding box of vertices. This update is divided into eight parts, one for each side of the box updating either the horizontal spokes or the vertical separator. The division avoids conflict with neighboring subproblems for access to the processors of the box, as in the factorization. The algorithm then follows with the backward solve of the cross.

We improve on Gannon's algorithm by again mimicking the factorization algorithm. Start by updating the vertical separator vertices with respect to the four sides of the enclosing box. Each part consists of loading the interior processors with elements from L^t, then calculating the update in a fashion similar to that in the forward solve algorithm. The results are not returned to the vertical separator processors between updates by the four sides. The vertical separator is then backsolved using a triangular processor array as in the forward solve algorithm. When these results have been returned to the vertical separator processors the two horizontal spokes are updated in parallel by the four sides of the boxes they are enclosed in, one side at a time. The data movement is again analogous to that of the factorization. This step ends with each horizontal spoke being backsolved using a triangular processor array. Thus this backward solve algorithm imitates the factorization algorithm with the ordering of its basic steps reversed, just as the forward solve algorithm uses the same ordering as in the factorization.

Analyzing this variant of Gannon's backward solve gives:

Theorem 6.3. *A variation of Gannon's backward solve algorithm on this architecture takes* $21n + (10r - 32)\log_2(n+1) - 13r - 8$ *forward solve steps and* $17n + (11r - 31)\log_2(n+1) - 20r - 3$ *communication steps.*

This backward solve algorithm requires a more complex program, and this

would effect its running time, but for multiple right hand sides it soon becomes cost effective.

7. Practicalities. In the previous two sections we gave detailed operation counts for the algorithm. The constants in the operation counts can be improved to some degree but at the cost of increased program overhead, program size, and programming difficulty. We are still working out the program specifics, but it is clear that the program size will be independent of n, and that the same code can reside in all processors.

We assumed that every processor along one side of the processor array can communicate with something external to the processor array. For these communication lines to be used in parallel requires that the processor array be communicating with another parallel processor capable of n simultaneous reads or writes, or with a serial processor with a communication channel which is enough faster than the processor array's cycle time that it appears to be a parallel processor (for some fixed n). For the program load it doesn't matter whether it is loaded in parallel through n processors, or whether it is loaded into one processor to start with. If the program consists of m words it will take $m+n-1$ communication steps to load the processor array given parallel external communication, and it will take at most $2m+2n-3$ communication steps if only one processor can communicate externally.

The data load sends 9 floating point numbers to each processor, representing the nonzeroes in each column of A. Another floating point number per processor is loaded representing $\bar{b}$. If parallel external communication is available the loading the matrix requires $n+8$ communication steps, and loading the right hand side requires n communication steps. If only one processor can communicate externally the data load degrades to $9n^2$ communication steps for loading the matrix and n^2 for loading $\bar{b}$. This dependence on n^2 will not dominate the cost of solving $A\bar{x}=\bar{b}$ until n is larger than $260+16/\rho$, where ρ is the ratio of the cost of interprocessor communication to floating point arithmetic. Another related problem is that the calculation of the entries of A and $\bar{b}$ can dominate the cost of solving $A\bar{x}=\bar{b}$ for small to moderate n even in serial machines. While the speed up of any part of this process is not to be ignored, for significant savings the generation of the matrix and the right hand side may need to be moved onto the processor array.

The retrieval of the solution $\bar{x}$ costs n communication steps given parallel external communication, and n^2 communication steps if the data is funneled through one processor. This bottleneck is not avoidable. If the elements of the matrix factorization are desired even the assumed parallel external communication is not sufficient. L contains $O(n^2\log_2(n+1))$ nonzeroes [3], with $O(n)$ nonzeroes in each processor on the level l cross. Removing this information requires $O(n\log_2(n+1))$ communication steps.

8. Conclusions and Comparisons. From sections 5 and 6 we can cal-

culate the total amount of work required to solve $A\bar{x} = \bar{b}$. Assume that interprocessor communication within the factorization and triangular solve steps is no cheaper than the cost within a pure communication step. Assume that two multiplications and one addition cost more than one division. We define synchronization logic and other housekeeping duties to be *step overhead.* This gives us

Theorem 8.1. *Asymptotic costs for solving $A\bar{x} = \bar{b}$ are*

$[126 + 172\rho + 109\sigma]n + O(\log_2(n+1))$	*for the factorization*
$[22 + 61.5\rho + 39.5\sigma]n + O(\log_2(n+1))$	*for the forward solve*
2	*for the diagonal solve*
$[10 + 30\rho + 10\sigma]n + O(\log_2(n+1))$	*for the backward solve*

Each step represents a floating point multiplication and a floating point addition or subtraction. σ is the relative cost of the step overhead compared to the combined cost of the multiplication and the addition, and ρ is the relative cost of a write and a read compared to the cost of a multiplication and an addition.

Proof: Simple arithmetic using Theorems 5.1, 6.1, and 6.2 suffices to give the result. This result is an overestimate due to the conservative combination of multiplications with additions and writes with reads. ∎

A similar analysis for multiple right hand sides is straight forward, where now the alternative backward solve algorithm is used.

Corollary 8.1. *The asymptotic efficiency E of this algorithm for solving $A\bar{x} = \bar{b}$ on this architecture can be bounded.*

$E_{fs} \in \left(\frac{1}{46}, \frac{1}{12}\right)$	*for the factorisation*
$E_{fs} \in \left(\frac{2}{31}\frac{\log_2(n+1)}{n}, \frac{4}{11}\frac{\log_2(n+1)}{n}\right)$	*for the forward solve*
$E_{ds} = 1$	*for the diagonal solve*
$E_{bs} \in \left(\frac{4}{15}\frac{\log_2(n+1)}{n}, \frac{4}{5}\frac{\log_2(n+1)}{n}\right)$	*for the backward solve*

For the solution using the separate forward solve algorithm,

$$E_s \in \left(\frac{1}{63}, \frac{1}{15}\right).$$

For the solution using the combined factorisation/forward solve algorithm,

$$E_s \in \left(\frac{1}{51}, \frac{1}{13}\right).$$

Proof: We define the efficiency of a parallel algorithm to be

$$E = \frac{\dfrac{\text{Complexity of serial algorithm}}{\text{Complexity of parallel algorithm}}}{\text{Number of processors}}.$$

This is an approximation to the usual definition of efficiency [7] which compares the time spent in executing the serial and parallel versions of the algorithm rather than their complexities. For example, we ignore the memory access times in both the serial and parallel algorithms. It is reasonable to assume that $\rho, \sigma \in (0,1)$, which provides the upper and lower bounds for the efficiency estimates. The exact values will depend on the architecture. The serial complexity of the factorization is

$$\frac{267}{28}n^3 + O(n^2 \log_2 (n+1))$$

and the serial complexity of the triangular solves is

$$\frac{93}{12}n^2 \log_2 (n+1) + O(n^2)$$

[3]. The results then follow from Theorem 8.1. ∎

We achieve linear speedup* for the solution of $A\bar{x} = \bar{b}$, although the triangular matrix solves are not as parallelizable with this algorithm and architecture. We expect σ and ρ to be fairly small and expect the upper bounds on the efficiencies in Corollary 8.1 to be good approximations. While this algorithm provides linear speedup, on the average less than one twelfth of the processors are working at any one time.

This loss of efficiency is due to two factors. We partition a k^{th} level subproblem into small enough parts to be solvable using only the local idle processors, and we synchronize the execution of these parts with neighboring subproblems to avoid conflicts of access to the shared processors. The division of the subproblem into parts influences the efficiency by increasing the overhead incurred in filling and emptying the pipelines of the systolic algorithms. It also incurs a cost in pure communication steps to arrange the data before and after each part. The synchronization cost stems from similar parts in different subproblems requiring different amounts of work. The most expensive case is assumed to insure no conflict between subproblems. There is also a percentage of loss due to the interprocessor communication associated with each step involving arithmetic. This is assumed to be more expensive than the corresponding memory access which occurs in the serial version.

* The speedup is the efficiency times the number of processors being used. The speedup is linear (in the number of processors) if the efficiency is independent of the number of processors.

The parallel algorithm is not obviously faster than the serial algorithm for small n. The parallel algorithm spends time in unproductive work in the pure communication steps and the synchronization between subproblems. The asymptotic analysis implies that the parallel algorithm is faster for $n \geq 7$ if both ρ and σ are less than .7. But for small n the waste is less and the lower order terms become more important. For small ρ and σ the parallel algorithm is faster for n as small as 3.

A competing parallel algorithm is the Kung–Leiserson systolic factorisation algorithm for banded dense matrices [9]. This systolic architecture requires n^2 processors and the algorithm finishes in $3n^2+n$ steps since the minimal bandwidth for the model problem is n [3]. Each step consists of three writes, three reads, two multiplications, and one addition. This is close to the basic step in our algorithm. From the asymptotic analysis we see that the parallel nested dissection algorithm will be faster for $n \geq 31$. The processors for the Kung–Leiserson systolic array do not need much local memory, but the systolic array must be fed data in parallel from an external source.

In conclusion, this algorithm efficiently parallelises the nested dissection algorithm using a simple MIMD architecture. The architecture allows linear speedup of what is a near optimal serial algorithm. Likewise, the parallel algorithm solves a problem which is globally dependent on its data in the same order of time it takes to communicate a single datum from one side of the processor grid to the other. Thus this architecture and algorithm are well matched for solving the model problem. As larger multiprocessors are built this algorithm, and its variations for more general problems, should be useful when a direct factorisation is desired to solve problems similar to the model problem.

REFERENCES

[1] G. BIRKHOFF and J. A. GEORGE, *Elimination by nested dissection*, in Complexity of Sequential and Parallel Numerical Algorithms, J. F. Traub, ed., Academic, New York, 1973.

[2] D. GANNON, *A note on pipelining a mesh connected multiprocessor for finite element problems by nested dissection*, in Proceedings of the 1980 Intl. Conf. on Parallel Processing (August, 1980), IEEE Catalog No. 80CH1569-3, pp. 197–204.

[3] A. GEORGE *Nested dissection of a regular finite element mesh*, SIAM J. Numer. Anal. 10 (1973), 345–363.

[4] A. GEORGE and J. W. LIU, *Computer Solution of Large Sparse Positive Definite Systems*, Prentice–Hall, Englewood Cliffs, N. J., 1981.

[5] R. W. HOCKNEY and C. R. JESSHOPE, *Parallel Computers*, Adam Hilger Ltd., Bristol, 1981.

[6] A. J. HOFFMAN, M. S. MARTIN, and D. J. ROSE, *Complexity bounds for regular finite difference and finite element grids*, SIAM J. Numer. Anal. 10 (1973), pp. 364–369.

[7] D. J. KUCK, *The Structure of Computers and Computation, Vol. 1*, John Wiley & Sons, Inc., New York, 1978.

[8] H. T. KUNG *Systolic algorithms*, in Large Scale Scientific Computation, S. Parter, ed., Academic Press, Inc., Orlando, FL, 1984.

[9] H. T. KUNG and C. E. LEISERSON, *Algorithms for VLSI processor arrays*, in Introduction to VLSI Systems, C. Mead and L. Conway, Addison-Wesley, Reading, MA, 1980.

[10] J. W. H. LIU, *The solution of mesh equations on a parallel computer*, Report CS-78-19 (Oct. 1978), Dept. of Computer Science, Univ. Waterloo, Waterloo, Ontario.

[11] P. H. WORLEY and R. SCHREIBER, *A parallel implementation of the nested dissection algorithm for elliptic PDEs*, Report CLaSSiC-85-03, Center for Large Scale Scientific Computation, Dept. of Computer Science, Stanford University, Stanford, CA. (*in preparation*)

END

FILMED

6-85

DTIC